601.465/665 — Natural Language Processing
Homework 2: Probability and Vector Exercises

Prof. Jason Eisner — Fall 2024
Due date: Mon 23 September, 2 pm

Only a little programming is required for this homework. The programming will get you started on some concepts and installation issues you'll need for the next homework. Mostly, you'll solve some pencil-and-paper problems, and then work through a long online visualization and complete a short program. There will be a **short quiz** about the visualization in the class immediately after this homework is due.

Homework goals: Completing this homework should make you feel comfortable with using probability expressions, Bayes' Theorem, n-gram models and similar models of word sequences, and log-linear models. You will also get some initial exposure to word embeddings and to the PyTorch library.

Collaboration: ***You may discuss each problem with one other student in the class.*** However, please write up your answers separately and hand them in separately. Otherwise it's too easy (for this homework) to get by without understanding. For the coding task, write your own program since it's quite short. If you discuss a problem with someone else, disclose this in your PDF and mention your partner's name.
You may not discuss with your partners from HW1. Make new friends! :-)

Starter code: There is some starter code available for question 8(a). See that question for the link.

How to hand in your work: Put all notes, documentation, and answers to questions in a PDF file, and submit it via Gradescope. You will submit your code separately.

The ✎ symbol in the left margin marks items that you need to hand in. In your PDF, you should refer to question numbers like 3(a). (Don't use the blue ✎ numbers; they are for your personal reference but may change if this homework handout is updated.)

Notation: When you are writing your PDF file, you will need some way of typing mathematical symbols. A good option is to use LaTeX. Alternatively, you could try the equation editor in Google Docs, Microsoft Word, or another word processor, or you could embed scans or photos of your neatly handwritten equations.

If you must type plain text, please pick one of the following three notations and use it consistently throughout your homework. (If you need some additional notation not described here, just describe it clearly and use it.) Use parentheses as needed to disambiguate division and other operators.

	Text	Picts	LaTeX
$p(x \mid y)$	`p(x | y)`	`p(x | y)`	`p(x \mid y)`
$\neg x$	`NOT x`	`~x`	`\neg x`
$\bar{x}$ (set complement)	`COMPL(x)`	`\x`	`\bar{x}`
$x \subseteq y$	`x SUBSET y`	`x {= y`	`x \subseteq y`
$x \supseteq y$	`x SUPERSET y`	`x }= y`	`x \supseteq y`
$x \cup y$	`x UNION y`	`x U y`	`x \cup y`
$x \cap y$	`x INTERSECT y`	`x ^ y`	`x \cap y`
$x \geq y$	`x GREATEREQ y`	`x >= y`	`x \geq y`
$x \leq y$	`x LESSEQ y`	`x <= y`	`x \leq y`
$\emptyset$ (empty set)	`NULL`	`0`	`\emptyset`
$\mathcal{E}$ (event space)	`E`	`E`	`E`

1. These short problems will help you get the hang of manipulating probabilities. Let $\mathcal{E} \neq \emptyset$ denote the event space (it's just a set, also known as the outcome space or sample space), and p be a function that assigns a real number in $[0, 1]$ to any subset of $\mathcal{E}$. This number is called the probability of the subset.

   You are told that p satisfies the following two axioms: $p(\mathcal{E}) = 1$. $p(X \cup Y) = p(X) + p(Y)$ provided that $X \cap Y = \emptyset$.[1]

   As a matter of notation, remember that the **conditional probability** $p(X \mid Z) \stackrel{\text{def}}{=} \frac{p(X \cap Z)}{p(Z)}$. For example, singing in the rain is one of my favorite rainy-day activities: so my ratio $p(\text{singing} \mid \text{rainy}) = \frac{p(\text{singing AND rainy})}{p(\text{rainy})}$ is high. Here the predicate "singing" picks out the set of singing events in $\mathcal{E}$, "rainy" picks out the set of rainy events, and the conjoined predicate "singing AND rainy" picks out the intersection of these two sets—that is, all events that are both singing AND rainy.

   (a) Prove from the axioms that if $Y \subseteq Z$, then $p(Y) \leq p(Z)$.
   You may use any and all set manipulations you like. Remember that $p(A) = 0$ does not imply that $A = \emptyset$ (why not?), and similarly, that $p(B) = p(C)$ does not imply that $B = C$ (even if $B \subseteq C$).

   (b) Use the above fact to prove that conditional probabilities $p(X \mid Z)$, just like ordinary probabilities, always fall in the range $[0, 1]$.

   (c) Prove from the axioms that $p(\emptyset) = 0$.

   (d) Let $\bar{X}$ denote $\mathcal{E} - X$. Prove from the axioms that $p(X) = 1 - p(\bar{X})$. For example, $p(\text{singing}) = 1 - p(\text{NOT singing})$.

   (e) Prove from the axioms that $p(\text{singing AND rainy} \mid \text{rainy}) = p(\text{singing} \mid \text{rainy})$.

   (f) Prove from the axioms that $p(X \mid Y) = 1 - p(\bar{X} \mid Y)$. For example, $p(\text{singing} \mid \text{rainy}) = 1 - p(\text{NOT singing} \mid \text{rainy})$. This is a generalization of question 1 (d).

   (g) Simplify: $\big(p(X \mid Y) \cdot p(Y) + p(X \mid \bar{Y}) \cdot p(\bar{Y})\big) \cdot p(\bar{Z} \mid X)/p(\bar{Z})$

   (h) Under what conditions is it true that $p(\text{singing OR rainy}) = p(\text{singing}) + p(\text{rainy})$?

   (i) Under what conditions is it true that $p(\text{singing AND rainy}) = p(\text{singing}) \cdot p(\text{rainy})$?

   (j) Suppose you know that $p(X \mid Y) = 0$. Prove that $p(X \mid Y, Z) = 0$.[2]

   (k) Suppose you know that $p(W \mid Y) = 1$. Prove that $p(W \mid Y, Z) = 1$.[3]

2. All cars are either red or blue. The witness claimed the car that hit the pedestrian was blue. Witnesses are believed to be about 80% reliable in reporting car color, regardless of the actual car color.[4] But only 10% of all cars are blue.

[1] In fact, probability functions p are also required to satisfy a generalization of this second axiom: if $X_1, X_2, X_3, \ldots$ is an infinite sequence of disjoint sets, then $p(\bigcup_{i=1}^{\infty} X_i) = \sum_{i=1}^{\infty} p(X_i)$. But you don't need this for this homework.

[2] More precisely, $p(X \mid Y, Z)$ could be either 0 or undefined, namely 0/0. (There do exist advanced ways to redefine conditional probability to avoid this 0/0 problem. Even then, though, one may want a probability measure p to leave some probabilities or conditional probabilities undefined. This turns out to be important for reasons beyond the scope of this course: e.g. http://en.wikipedia.org/wiki/Vitali_set.)

[3] More precisely, $p(X \mid Y, Z)$ could be either 1 or undefined. See the previous footnote.

[4] In other words, accuracy is independent of color: 80% of witnesses who see red cars report the color correctly, and so do 80% of witnesses who see blue cars.

(a) Write an equation relating the following quantities and perhaps other quantities:

$$p(\textit{Actual} = \text{blue})$$
$$p(\textit{Actual} = \text{blue} \mid \textit{Claimed} = \text{blue})$$
$$p(\textit{Claimed} = \text{blue} \mid \textit{Actual} = \text{blue})$$

Reminder: Here, *Claimed* and *Actual* are *random variables*, which means that they are functions over some outcome space. For example, the probability that *Claimed* = blue really means the probability of getting an outcome x such that $\textit{Claimed}(x) = \text{blue}$. We are implicitly assuming that the space of outcomes x is something like the set of witnessed car accidents.

(b) Match the events *Actual* = blue and *Claimed* = blue with the following terms:

the *hypothesis* you are evaluating
the *evidence* you have observed

Similarly, match the 3 probabilities in question 2(a) with the following terms:

prior probability of the hypothesis
likelihood of the hypothesis
posterior probability of the hypothesis

(c) Give the values of all three probabilities in question 2(a). (Hint: Use Bayes' Theorem.) Which probability should the judge care about?

(d) Let's suppose the numbers 80% and 10% are specific to Baltimore. So in the previous problem, you were implicitly using the following more general version of Bayes' Theorem:

$$p(A \mid B, Y) = \frac{p(B \mid A, Y) \cdot p(A \mid Y)}{p(B \mid Y)}$$

where Y is *city* = Baltimore. Just as question 1(f) generalized question 1(d), by adding a "background" condition Y, this version generalizes Bayes' Theorem. Carefully prove it.

(e) Now prove the more detailed version

$$p(A \mid B, Y) = \frac{p(B \mid A, Y) \cdot p(A \mid Y)}{p(B \mid A, Y) \cdot p(A \mid Y) + p(B \mid \bar{A}, Y) \cdot p(\bar{A} \mid Y)}$$

which gives a practical way of finding the denominator in question 2(d).

(f) Write out the equation given in question 2(e) with A, B, and Y replaced by specific propositions from the red-and-blue car problem. For example, Y is "*city* = Baltimore" (or just "Baltimore" for short). Now replace the probabilities with actual numbers from the problem, such as 0.8.

Yeah, it's a mickeymouse problem, but I promise that writing out a real case of this important formula won't kill you, and may even be good for you (like, on an exam).

3. Beavers can make three cries, which they use to communicate. `bwa` and `bwee` usually mean something like "come" and "go" respectively, and are used during dam maintenance. `kiki` means "watch out!" The following **conditional probability table** shows the probability of the various cries in different situations.

$p(cry \mid situation)$	Predator!	Timber!	I need help!
`bwa`	0	0.1	0.8
`bwee`	0	0.6	0.1
`kiki`	1.0	0.3	0.1

(a) Notice that each column of the above table sums to 1. Write an equation stating this, in the form $\sum_{variable} p(\cdots) = 1$.

(b) A certain colony of beavers has already cut down all the trees around their dam. As there are no more to chew, $p(timber) = 0$. Getting rid of the trees has also reduced $p(predator)$ to 0.2. These facts are shown in the following **joint probability table**. Fill in the rest of the table, using the previous table and the laws of probability. (Note that the meaning of each table is given in its top left cell.)

$p(cry, situation)$	Predator!	Timber!	I need help!	TOTAL
`bwa`				
`bwee`				
`kiki`				
TOTAL	0.2	0		

(c) A beaver in this colony cries `kiki`. Given this cry, other beavers try to figure out the probability that there is a predator.

i. This probability is written as: $p(\_\_\_\_\_)$

ii. It can be rewritten without the $\mid$ symbol as: _____

iii. Using the above tables, its value is: _____

iv. Alternatively, Bayes' Theorem allows you to express this probability as:

$$\frac{p(\_\_\_\_\_) \cdot p(\_\_\_\_\_)}{p(\_\_\_\_\_) \cdot p(\_\_\_\_\_) + p(\_\_\_\_\_) \cdot p(\_\_\_\_\_) + p(\_\_\_\_\_) \cdot p(\_\_\_\_\_)}$$

v. Using the above tables, the value of this is:

$$\frac{\_\_\_\_\_ \cdot \_\_\_\_\_}{\_\_\_\_\_ \cdot \_\_\_\_\_ + \_\_\_\_\_ \cdot \_\_\_\_\_ + \_\_\_\_\_ \cdot \_\_\_\_\_}$$

This should give the same result as in part iii., and it should be clear that they are really the same computation—by constructing table (b) and doing part iii., you were *implicitly* using Bayes' Theorem. (I told you it was a trivial theorem!)

4. A **language model** is a probability function p that assigns probabilities to word sequences such as $\vec{w} = (\texttt{i}, \texttt{love}, \texttt{bagpipe}, \texttt{music})$.

Suppose $\vec{w} = w_1 w_2 \cdots w_n$ (a sequence of n words). A **trigram language model** defines

$$p(\vec{w}) \stackrel{\text{def}}{=} \prod_{i=1}^{n+1} p(w_i \mid w_{i-2}, w_{i-1}) \tag{1}$$

where by convention we define $w_{n+1} = \text{EOS}$ ("end of sequence") and $w_0 = w_{-1} = \text{BOS}$ ("beginning of sequence").

For example, the model says that a speaker who says "`i love bagpipe music`" has generated the sequence in left-to-right order, by rolling 5 dice that happened to land on `i`, `love`, `bagpipe`, `music`, EOS. Each roll uses a special die selected by the two previously rolled characters (or BOS).

(a) Explicitly write out $p(w_1 w_2 w_3 w_4)$ when you use naive estimates of the parameters, such as

$$p(w_4 \mid w_2, w_3) \stackrel{\text{def}}{=} \frac{c(w_2 w_3 w_4)}{c(w_2 w_3)} \tag{2}$$

where $c(w_2 w_3 w_4)$ denotes the *count* of times the trigram $w_2 w_3 w_4$ was observed in a training corpus. The following terms will appear in your formula: $c(\text{BOS BOS})$, $c(\text{BOS BOS } \texttt{i})$, $c(\texttt{bagpipe music EOS})$. What property of the corpus is counted by each of them?

Remark: Naive parameter estimates of this sort are called **maximum-likelihood estimates** (MLE). They have the advantage that they maximize the probability (equivalently, minimize the perplexity) of the training data. But they will generally perform badly on test data, unless the training data were so abundant as to include all possible trigrams many times. This is why we must smooth these estimates in practice.

(b) Suppose $\vec{w} = (\texttt{do}, \texttt{you}, \texttt{think}, \texttt{the})$. Under any good language model of English, $p(\vec{w})$ should be extremely low. Why? In the case of the trigram model, which parameter or parameters are responsible for making this probability low?

(c) You turn on the radio while it is in the middle of broadcasting an interview. Assume that each sentence of the interview was independently generated by a certain trigram model.

Under this assumption, match up expressions (A), (B), (C) with descriptions (1), (2), (3):

The expression

(A) $p(\texttt{do}) \cdot p(\texttt{you} \mid \texttt{do}) \cdot p(\texttt{think} \mid \texttt{do}, \texttt{you})$

(B) $p(\texttt{do} \mid \text{BOS}) \cdot p(\texttt{you} \mid \text{BOS}, \texttt{do}) \cdot p(\texttt{think} \mid \texttt{do}, \texttt{you}) \cdot p(\text{EOS} \mid \texttt{you}, \texttt{think})$

(C) $p(\texttt{do} \mid \text{BOS}) \cdot p(\texttt{you} \mid \text{BOS}, \texttt{do}) \cdot p(\texttt{think} \mid \texttt{do}, \texttt{you})$

represents the probability that

(1) the first complete sentence you hear is `do you think`

(2) the first 3 words you hear are `do you think`, all as part of a single sentence

(3) the first complete sentence you hear starts with `do you think`

Explain your answers briefly. *Remark:* The distinctions matter because "do" is more probable at the start of an English sentence than in the middle, and because (3) describes a larger event set than (1) does. *Remark:* The best reply to `do you think` is "Yes, and therefore I am!"

(d) **Extra credit:** One could also define a kind of reversed trigram language model p_{reversed} that instead assumed the words were generated in reverse order ("from right to left"):

$$p_{\text{reversed}}(\vec{w}) \stackrel{\text{def}}{=} \prod_{i=0}^{n} p(w_i \mid w_{i+1}, w_{i+2}) \tag{3}$$

where by convention we define $w_0 = \text{BOS}$ and $w_{n+1} = w_{n+2} = \text{EOS}$. In this case, the generating process stops when we hit BOS, instead of EOS.

By manipulating the notation, show that the two models are identical (i.e., $p(\vec{w}) = p_{\text{reversed}}(\vec{w})$ for any $\vec{w}$) provided that both models use MLE parameters estimated from the same training data (see question 4(a)).

Hint: Try writing out the probability of "`i love bagpipe music`" under both models. To argue that the resulting probabilities are equal, you will have to observe that certain counts are equal.

5. Under a bigram language model, knowing an early part of the sentence (*"Horses like . . . "*) doesn't tell you too much about the end of the sentence. But that overlooks a useful property of real language data. A sentence that starts out *"Horses like . . . "* is probably about horses, so it will probably continue to have a lot of horse-related words throughout, even at the end.

   Suppose your corpus seems to cover k different topics—for example, POLITICS, CELEBRITIES, ANIMALS, and SPORTS. For the most part, each sentence seems to stick to a single topic. **You want to build an better bigram model that also captures the fact the choice of topic persists throughout each sentence.**

   In the "probability crash course" lecture, we sketched a model of word sequences that considered possible part-of-speech tags for the first word. The goal wasn't to predict tags, but since words are related to tags, we hoped that a model that included tags might predict the word sequence better. In the same way, the goal here is not to predict topics, but to use them to predict the word sequence better.

   Write a formula for $p(w_1 w_2 w_3 w_4)$ under this better bigram model. (*Hints:* Latent variable, chain rule, backing off using a conditional independence assumption. Make w_4 depend not only on w_3 but also on the topic a.

6. **[This problem is very important to the course, and will take you several hours to do properly.]**

   So far in this homework, we have been using conditional probabilities like $p(\textit{Actual} = \text{blue})$ or $p(\text{singing} \mid \text{rainy})$ or $p(w_4 \mid w_2, w_3)$. But where do those probabilities come from?

   One option would be to estimate conditional event probabilities naively from data, using simple count ratios as in formula (2). But those unsmoothed estimates will not be reliable when the numerator or denominator counts are small.

   An alternative is to use log-linear models. Frank Ferraro and I built an online toy to let you play with such models. It should give you almost *physical* intuitions about how these models behave.

   You'll need to understand log-linear models in subsequent lectures and on the exams. They are also a pretty good introduction to machine learning.

   You can find the toy at `http://cs.jhu.edu/~jason/tutorials/loglin/`. Work through its 18 lessons (and read the accompanying handout). This should be fun, but time-consuming.

   To save you some time, there is nothing to hand in for this part of the homework. **Instead, there will**
   

   be a brief but tricky in-class quiz to check that you understood what's going on. Your score on the quiz will be your score for this homework question.

   The lessons are peppered with guidance in the form of questions. Just think through these questions as you go along, and you should do fine on the quiz. We strongly suggest that you **work through the lessons with a friend or two**, so that you can discuss the questions together.

7. Now, how might you use log-linear models in linguistics? Rather than answering in your PDF, do one of the following:

   - ✍8 Think over your own interests in linguistics, NLP, or a related field. Post a new note on Piazza (a note, not a question) with a subject line "Log-linear model for [my idea]." While we generally allow you to post anonymously, please don't for this question—it means we can't search by name, which slows down grading.

     In your note, give an example of a conditional distribution $p(y \mid x)$ that would be interesting to model. It could be related to natural language, but it doesn't have to be. Your goal here might be to predict y from x, or to understand the properties of x that are predictive of y.

     Be clear about both your real-world problem and your model of it. In particular, be precise about what the formal objects x and y are—numbers? strings? arrays? trees?—*and* what x and y represent in the real world. **We will post an "official example" on Piazza** to illustrate the form of a good answer.

   - ✍8 **Alternatively**, respond to someone else's Piazza post that you found interesting. Suggest some new features $f_k(x, y)$ that would be useful for their problem. Or suggest changes to the problem setup (e.g., the choice of x and y) or to other people's features. Again, please don't comment anonymously.

   Be precise when describing your feature functions. Don't just say "the amount of pseudoscience words used in the text." That's a certainly a good starting thought—but how should your program identify "pseudoscience words"? (Would you need to supply it with a list of such words, or could you somehow pluck them automatically from a corpus of pseudoscience articles?) And how exactly would your feature function measure this "amount"? (Should it return an integer count? A fraction? The log of a fraction?)

   Consider designing a large, systematic set of feature functions—hundreds or thousands that *might* help prediction, not just 3 or 4 that *should* help. Hopefully, optimizing the weights will find the features that *do* help and give weight ≈ 0 to the others. For example, instead of a single feature that fires whenever it sees a pseudoscience word, you could create separate features that fire for each word from "aardvark" to "zebra," and hope that the system will learn to put a positive weight on "megalithic" and a negative weight on "t-test."

   Warning: If you're trying to predict different probabilities for different y values, then your feature vector $\vec{f}(x, y)$ needs to be different for different y values, too. (Remember lesson 14 from the visualization.) Suppose you are given a text x and you're trying to predict its author's age y. If (for a given x) your features fire in the same way on all ages y, then all ages will have the same probability: $p(0 \mid x) = p(1 \mid x) = p(2 \mid x) = \ldots = p(99 \mid x) = \frac{1}{100}$. So don't use a feature like "$f_3(x, y) = 1$ if x contains an emoji." You will need a set of more specific features that depend on y, such as "$f_3(x, y) = 1$ if x contains an emoji and $y < 18$."[5]

8. The next homework will involve various kinds of smoothed language models. Often, smoothing involves treating similar events in similar contexts as having similar probabilities. (Backoff is one example.)

[5] If this feature has positive weight, then an emoji raises the probability that the author is a minor. Specifically, it raises the probabilities that $y = 0, y = 1, \ldots, y = 17$, which means lowering the probabilities that $y = 18, y = 19, \ldots$. The bad version that doesn't test y tries to raise the probabilities of all ages at once—but this has no effect after normalization: doubling all of the unnormalized probabilities doesn't change their ratios.

One strategy will be to treat similar *words* as having similar probabilities. But what does "similar words" mean? More concretely, how will we *measure* whether two words are similar?

Look at http://cs.jhu.edu/~jason/465/hw-lm/lexicons/.[6] Remember that a *lexicon* lists useful properties of the words of a language. Each of the `words-*.txt` files[7] is a lexicon that lists over 70,000 words of English, with a representation of each word as a d-dimensional vector. In other words, it *embeds* these words into the vector space $\mathbb{R}^d$; so the vectors are often called "word embeddings."

We can now define "similar words" as words with similar vectors. A word may have many syntactic and semantic properties—some words are transitive verbs, some words are plural, some words refer to animals. These properties can be considered by a log-linear model, and words with similar vectors tend to have similar properties. The larger d is, the more room the vector has to encode interesting properties.

The vectors in these particular files were produced automatically by Google's `word2vec` program.[8] Their individual dimensions are hard to interpret as natural properties. It would certainly be nice if dimension 2 (for example) represented the "animal" property, so that a word that referred to an animal would be one whose vector $\vec{v} = (v_1, v_2, \ldots v_d)$ had strongly positive v_2. In practice, however, `word2vec` chooses an arbitrary basis for the vector space. So the "animal" direction—to the extent that there is one—is actually represented by some arbitrary vector $\vec{u}$. The animal words tend to be those whose vectors $\vec{v}$ have strongly positive $\vec{v} \cdot \vec{u}$.

(Geometrically, this dot product measures how far $\vec{v}$ extends in direction $\vec{u}$ (it projects $\vec{v}$ onto the $\vec{u}$ vector). Algebraically, it computes a certain linear combination of $v_1, v_2, \ldots, v_n$. As a special case, if $\vec{u}$ were $(0, 1, 0, 0, \ldots)$, then $\vec{v} \cdot \vec{u}$ would in fact return v_2. But there's no reason to expect that $\vec{u}$ would be that simple.)

You'll be using our lexicons in the next homework. For this homework, you'll just warm up by writing a short program to get a feel for word embeddings, vectorized computation, and the PyTorch library. Start with the `INSTRUCTIONS` file in http://cs.jhu.edu/~jason/465/hw-prob/.

(a) Your program `findsim.py` should print the 10 words most similar to `seattle`, other than `seattle` itself, according to the 50-dimensional embeddings, if you run it as

```
python3 findsim.py words-50.txt seattle
```

They should be printed in decreasing order of similarity. To measure the similarity of two words with vectors $\vec{v}$ and $\vec{w}$, please use the *cosine similarity*, which is the cosine of the angle θ between

[6] If you lack the bandwidth or disk space to download all of these files to your own machine, just download a few. Once your code is working, you can upload your code to the `ugrad` filesystem and run it on one of the `ugrad` machines, where the files are available in the directory `/usr/local/data/cs465/hw-lm/lexicons/`. Please don't make additional copies on the `ugrad` filesystem, as this would waste space; you can use symbolic links instead.

[7] The file format should be easy to figure out. The file `words-`d`.txt` can be regarded as a matrix with about 70,000 rows and d columns. Each row is labeled by a word. The first line of the file is special: it gives the number of rows and columns.

[8] The details are not important for this homework, but the vectors are optimized by gradient descent so as to arrange that the vector for each word token w_i will be predictable (to the extent possible) from the average vector of nearby word tokens (w_j for $j \neq i$ and $i - 5 \leq j \leq i + 5$). If you're curious, you can find details in Mikolov et al. (2013)'s paper "Distributed Representations of Words and Phrases and their Compositionality," available at http://arxiv.org/abs/1301.3781. We ran the CBOW method over the first 1 billion characters of English Wikipedia. `word2vec` doesn't produce vector representations for rare words ($c(w) < 5$), so we first replaced rare words with the special symbol OOL ("out of lexicon"), forcing `word2vec` to learn a vector representation for OOL.

the two vectors:

$$\cos\theta = \left(\frac{\vec{v}}{||\vec{v}||}\right) \cdot \left(\frac{\vec{w}}{||\vec{w}||}\right) = \frac{\vec{v}\cdot\vec{w}}{||\vec{v}||\ ||\vec{w}||} = \frac{\sum_{i=1}^{d} v_i w_i}{\sqrt{\sum_{i=1}^{d} v_i^2}\sqrt{\sum_{i=1}^{d} w_i^2}} \in [-1, 1]$$

We've provided starter code for `findsim.py` in the directory mentioned above.

What are the most similar words to `seattle`, `dog`, `communist`, `jpg`, `the`, and `google`? Play with your program some more. What are some examples that work "well" or "poorly"? What patterns do you notice?

So far you have been using $d = 50$. What happens for larger or smaller values of d?

Hint: It's probably a good idea to read the lexicon into some data structures. (See remarks below.)

Hint: Start by making `findsim` find only the single most similar word, which should be easy enough. Then you can generalize this method to find the 10 most similar words. (Since you only want the top 10, it's not necessary to sort the whole lexicon by similarity—that method is acceptable, but inefficient.)

Note: You can copy the lexicon files to your personal machine. But to avoid downloading such large files, it may be easier to run `findsim` directly on the `ugrad` machines. If you do this, please **don't waste space** by making fresh copies of the files on the `ugrad` machines. Just use them at their current location. If you like, create symbolic links (shortcuts) to them by typing "`ln -s /usr/local/data/cs465/hw-lm/lexicons/* .`" in your own working directory.

Just for fun: Semantle is a game based on similar lexicons of `word2vec` embeddings. It tells you how close your guess is to the target word, measuring that by cosine similarity (whereas Wordle measures it by how many of the letters match). It takes a lot of guesses and the ability to think of lots of words! Try it if you like, perhaps starting with Semantle Junior, which uses easier target words.

(b) Now extend your program so that it can also be run as follows:

```
python3 findsim.py words-50.txt king --minus man --plus woman
```

This should find the 10 words most similar to the vector `king` − `man` + `woman`, other than those three words themselves.

(The old command `python3 findsim.py words-50.txt seattle` should still work. Note that it is equivalent to `python3 findsim.py words-50.txt seattle --minus seattle --plus seattle`, and you may want to implement it that way.)

You can regard the above command as completing the analogy

```
man : woman :: king : ?
```

Try some more analogies, this time using the **200-dimensional vectors**. For example:

```
king - man + woman
paris - france + uk
hitler - germany + italy
child - goose + geese
goes - eats + ate
car - road + air
```

☞12 Come up with some analogy questions of your own. Which ones work well or poorly? What happens to the results if you use (say) $d = 10$ instead of $d = 200$?

☞13 Why does this work at all? Be sure to discuss the role of the vector `king` − `man` before `woman` is added. What does it tell you about the vectors that they can solve analogies?

☞14 Submit your extended `findsim` program on Gradescope.

Data structures, PyTorch, and embeddings. Critical to data-driven methods in NLP is writing efficient code to handle large amounts of data. Neural networks moved to the center of the field around 2015. While neural nets were popular in AI in the 1960's and again in the late 1980's (your prof even did his undergrad thesis in this area), a third wave of interest grew out of impressive results that started arriving around 2006, and neural nets now seem to be here to stay. The availability of massive training data and parallel hardware—plus a bag of contemporary tricks for getting neural networks to learn—lets us perform computational feats that would have been impossible even a decade ago.

Excellent software frameworks have arisen for building and training neural networks. In this class, we'll introduce you to one called PyTorch. PyTorch is fundamentally a library for fast, space-efficient vectorized math in Python.[9] It's similar to the widespread math library NumPy in that regard, but PyTorch provides three advantages:

- PyTorch's mathematical objects track the computations applied to them, explicitly storing a computation graph and tracking the gradients of the computations. This is used for backpropagation, a key algorithm that tells us how to gradually improve the parameters of a neural network.
- PyTorch provides useful classes and functions built on top of its mathematical objects. These make it easy to build common kinds of neural networks and train them using standard algorithms.
- PyTorch's libraries make good use of parallel hardware. Modern CPUs provide vectorized operations. Better yet, if your computer has a GPU, then PyTorch can use it for incredibly fast parallel computation. (Our autograders don't have GPUs, so we won't turn to this for our class.)

Some of these strengths will come into play in the *next* homework. The *current* homework is just a PyTorch warmup, where you'll install the libraries and try out a little vectorized computation.

In order to find the words most similar to your query word (e.g. `seattle`), you'll have to compute the similarity between its embedding (a vector) and the other words' embeddings. One approach is to do this with a loop, comparing to each other word separately. But there's a better, faster way. We can *batch* the computations together, letting us compute the similarities in parallel.

Working out the details of this will be part of your task in this homework, but we'll provide some guidance here. Instead of making a Python dictionary that maps word types to vectors, you'll want to store all of the vectors together in a single matrix (see footnote 7). Each row of the matrix represents the embedding of a different word type. Packing all these vectors into a single matrix often allows you to run a computation in parallel over the whole vocabulary by using a single efficient PyTorch matrix operation. Modern hardware is designed for this setting, where it can easily load a big block of the matrix into the memory cache or GPU and run operations on its elements in parallel.

If you have a vocabulary of size V, the matrix will have V rows. You'll need to "integerize" the word types, identifying them with the integers $0, 1, 2, \ldots, V-1$. For example, mapping `seattle` to its

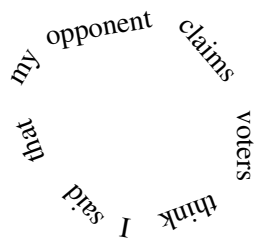
[9] A "vectorized" sin function means that if $\vec{v} = (v_0, \ldots, v_{49}) \in \mathbb{R}^{50}$, then $\sin(\vec{v})$ returns $(\sin(v_0), \ldots, \sin(v_{49})) \in \mathbb{R}^{50}$. PyTorch's implementation of vectorized sin is far faster than an equivalent Python loop, since Python is a slow interpreted language.

integer—suppose it's 57—will let you look up its embedding in row 57. If you find that row 99 is the most similar row, you'll have to map the integer 99 back to its spelling in order to print out the answer. We've provided an `Integerizer` class to maintain this two-way mapping.

Integerization is an example of the flyweight design pattern. Unlike strings, integers are small fixed-size objects: you can store a sentence compactly as an array of integers, and comparing two integers is much faster than comparing two strings. If we represent the word type `seattle` as the integer 57, we can use 57 as an index into arrays, vectors, and matrices that store various information about that word type, such as its spelling, embedding, pronunciation, syntactic category, meaning, corpus count, and token positions.[10] Of course, the number 57 is arbitrary: as XKCD points out, its numeric value tells you nothing about the word.

In this homework, a lexicon will only contain one instance of `seattle`. But more generally, a lexicon could include multiple word types that happen to have the same spelling. Chief Seattle was a Native American leader; the City of Seattle was named after him; and the Seattle Mariners baseball team was named after the city. All three can be referred to as `seattle`, but arguably they are different word types that should have different embeddings. This is naturally allowed if distinct word types are represented by distinct integers (or pointers) with associated data, rather than by distinct strings.

9. **Extra credit:** Some politicians seem to speak in circles. Instead of uttering strings with a start and an end, they blow round linguistic objects out of their mouths, like smoke rings:



I wondered, how would we build a language model to describe the probability of such objects? Well, problems like this show up in computer vision, since images don't have a natural start and end either. Consider the similar problem of modeling the colors of these 4 pixels, using bigrams of adjacent pixel colors:

A	B
D	C

A well-known computer vision paper by J. Besag (1974) proposed something like this: approximate $p(A, B, C, D)$ by a product $p(A \mid B) \cdot p(B \mid C) \cdot p(C \mid D) \cdot p(D \mid A)$.

15 So here's the question. Can this approximation be justified by using the chain rule plus backoff? If so, show how. If not, fix it as best you can. Discuss.

[10] The object-oriented alternative would be to represent the word type not as an integer, but as a pointer to a `Word` object that stores all the information about `seattle`. This keeps all the information about a word in one place. But for many computations, it makes more efficient use of the hardware to store all the embeddings consecutively in one region of memory, all the spellings in another region, etc.

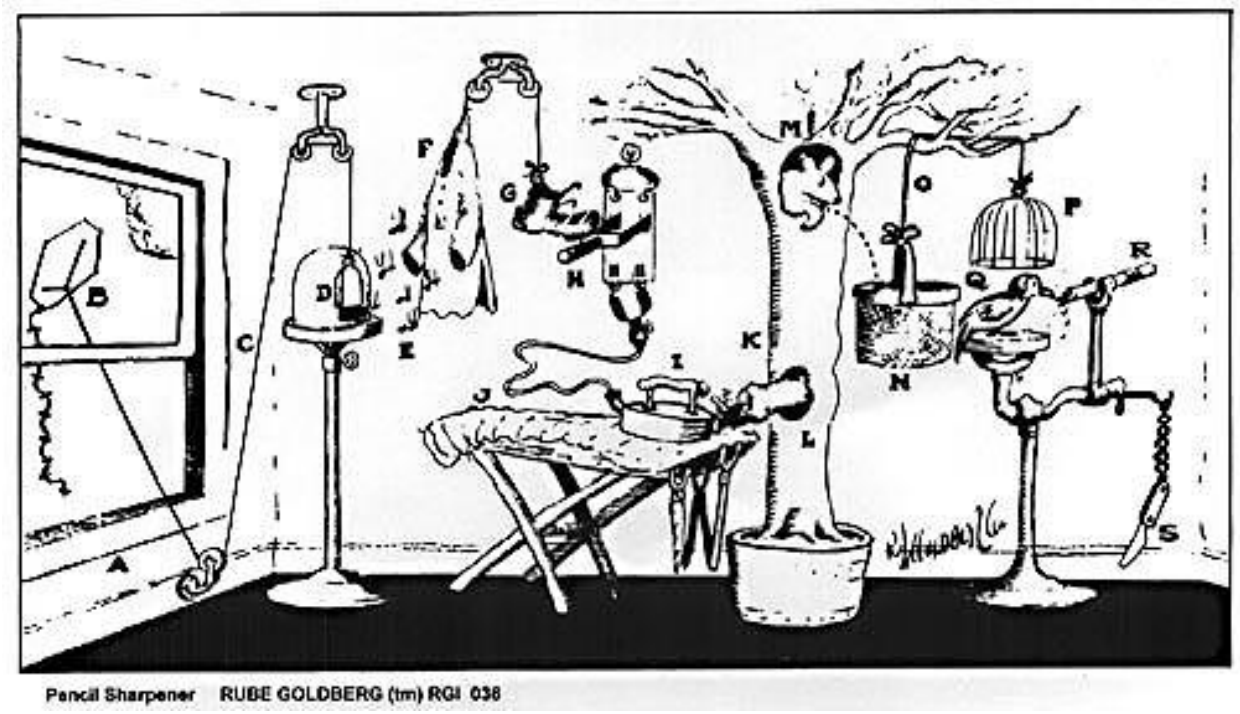


Figure 1: RUBE GOLDBERG GETS HIS THINK-TANK WORKING AND EVOLVES THE SIMPLIFIED PENCIL-SHARPENER. Open window (A) and fly kite (B). String (C) lifts small door (D) allowing moths (E) to escape and eat red flannel shirt (F). As weight of shirt becomes less, shoe (G) steps on switch (H) which heats electric iron (I) and burns hole in pants (J). Smoke (K) enters hole in tree (L), smoking out opossum (M) which jumps into basket (N), pulling rope (O) and lifting cage (P), allowing woodpecker (Q) to chew wood from pencil (R), exposing lead. Emergency knife (S) is always handy in case opossum or the woodpecker gets sick and can't work.

10. **Extra credit:** This problem is supposed to convince you that logical reasoning is is just a special case of probabilistic reasoning.[11] That is, probability theory allows you to draw a conclusion from some premises like "**If** there's no nail, **then** there's definitely no shoe."

(a)	$p(\neg\text{shoe} \mid \neg\text{nail}) = 1$	*For want of a nail the shoe was lost,*
(b)	$p(\neg\text{horse} \mid \neg\text{shoe}) = 1$	*For want of a shoe the horse was lost,*
(c)	$p(\neg\text{race} \mid \neg\text{horse}) = 1$	*For want of a horse the race was lost,*
(d)	$p(\neg\text{fortune} \mid \neg\text{race}) = 1$	*For want of a race the fortune was lost,*
(e)	$p(\neg\text{fortune} \mid \neg\text{nail}) = 1$	*And all for the want of a horseshoe nail.*

✌16 Show carefully that (e) follows from (a)–(d). *Hint:* Consider

$$p(\neg\text{fortune}, \neg\text{race}, \neg\text{horse}, \neg\text{shoe} \mid \neg\text{nail}),$$

as well as the "chain rule" and questions 1(a), 1(b) and 1(k).

Note: The $\neg$ symbol denotes the boolean operator NOT.

Note: Be glad I didn't ask you to prove the correct operation of Figure 1!

[11] An excellent and wide-ranging book developing this theme is *Probability Theory: The Logic of Science*, by the influential statistician E. T. Jaynes. See http://bayes.wustl.edu/ for more readings.